Artur Vasilian

# Assignment Report

**Link:** https://github.com/Archi-RK6/fork_exec_practice

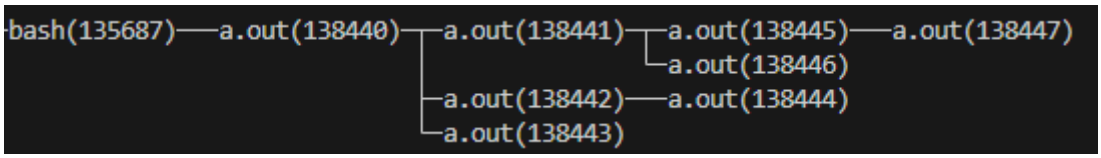
## Assignment 0: Multiple Fork calls

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    fork();
    fork();
    fork();

    printf("PID = %d parent PID = %d\n \n", getpid(), getppid());
    sleep(60);
    return 0;
}
```

The original process is the 138440. After first system call `fork()`, there are 2 processes (parent + 1 child 138441). After the second fork(), each of those can fork again and now we have 2 new processes (138442, 138445). After the third `fork()` all running processes created another child (138443, 138444, 138446, 138447). We have 8 processes in total.

Every created process executes the same code after each `fork()` call, so each of them prints its own PID and parent PID and then sleeps for 60 seconds, giving us time to inspect the process tree.

Process tree:




Output:

```
PID = 138440 parent PID = 135687
PID = 138444 parent PID = 138442
PID = 138442 parent PID = 138440
PID = 138441 parent PID = 138440
PID = 138443 parent PID = 138440
PID = 138445 parent PID = 138441
PID = 138446 parent PID = 138441
PID = 138447 parent PID = 138445
```

**Assignment 1: Simple Fork and Exec**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(){
    int ret = fork();

    if(ret == 0){
        execl("/bin/ls", "ls", NULL);
        perror("execl");
        exit(1);
    } else {
        wait(NULL);
        printf("Parent process done\n");
    }
    return 0;
}
```

After first `fork()` call the created child calls `execl` to replace running code with system call `ls` (which shows all files in current folder). Meanwhile parent process waits (using `wait(NULL)`) for child process to end running and then prints "Parent process done".

We distinguish in the code which process we are in due to the `ret` variable. `fork()` returns a value to it, and if we are in the child, then it is 0, if we are in parent it is not 0.

**Assignment 2: Multiple Forks and Execs**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(){
    int ret_1 = fork();

    if(ret_1 == 0){
        execl("/bin/ls", "ls", NULL);
        perror("execl 1");
        exit(1);
    }

    int ret_2 = fork();

    if(ret_2 == 0){
```

```
        execl("/bin/date", "date", NULL);
        perror("execl 2");
        exit(1);
    }

    wait(NULL);
    wait(NULL);

    printf("Parent process done\n");
    return 0;
}
```

After first `fork()` call the created child calls `execl` to replace running code with system call `ls`. Meanwhile parent continues running and creates another process. That new child process calls `execl` to replace running code with system call `date` (which outputs the current time and date). The parent then calls `wait()` twice to wait for both processes to end (a call to `wait()` blocks the calling process until one of its child processes exits or a signal is received).

Because the first child calls `execl` immediately, it doesn't execute the next `fork()`. Therefore, we have 2 children in total (one for ls, one for date).

**Assignment 3: Fork and Exec with Arguments**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(){
    int ret = fork();

    if(ret == 0){
          execl("/bin/echo", "echo", "Hello from the child process",
NULL);
        perror("execl 1");
        exit(1);
    }else{
        wait(NULL);
        printf("Parent process done\n");
    }

    return 0;
}
```

After first `fork()` call the created child calls `execl` to replace running code with system call `echo` with provided massage "Hello from the child process". And the parent process waits for child process to finish like in assignments before.

**Assignment 4: Fork and Exec with Command-Line Arguments**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(){
    int ret = fork();

    if (ret == 0){
        execl("/bin/grep", "grep", "test", "test.txt", NULL);
        perror("execl 1");
        exit(1);
    }else{
        wait(NULL);
        printf("Parent process completed\n");
    }

    return 0;
}
```

Parent creates a new child with `fork()` and the new process becomes `grep` command (which searches for a specific word in a text file) and prints all matching lines according to given parameters (what and where to find; “test” and “test.txt”). Parent waits for child process to end and prints it final massage “Parent process completed”.